Storage Client API Specification

DRAFT

Please send comments to:
{annc@isi.edu,foster@mcs.anl.gov,carl@isi.edu,salisbur@mcs.anl.gov,tuecke@mcs.anl.gov}

Abstract

We describe an application programmer interface (API) designed to support high-performance remote access to a variety of storage systems. This API is intended for use within distributed systems that support the integrated management and analysis of large (petabyte-scale) distributed data collections: what we term *Data Grids*. The definition of this API simplifies the implementation of Data Grid applications by providing a uniform interface to the diverse storage system types that may be encountered in Data Grid environments. The interface is defined so that implementations can exploit sophisticated techniques to achieve high performance, such as network striping, parallel I/O, and network protocol tuning. Hence, in principle, applications need not sacrifice performance for portability when using this API to move data between storage systems and either client applications or other storage systems.

Table of Contents

1 Background and Terminology

The Storage Client API (SC-API) that we define here is just one component of a larger Data Grid architecture. We provide a detailed description of the overall architecture of the grid elsewhere. In the subsection that follows, we describe some of the key components of the architecture, focusing on terminology.

The bulk of this document concerns the description of the SC-API. This API is concerned exclusively with *data access* and *data movement*. Programmers will typically use functions from the SC-API as well as from other Data Grid components (e.g., mechanisms for access to metadata catalogs, security, and computing resources) to implement a particular application.

1.1 Data Grid Components

A *logical file* is an entity that has a globally unique name and that may also have associated with it metadata and one or more physical *file instances.* A file instance is just an uninterpreted sequence of bytes which is located in a *storage system*. We propose to use X.500 distinguished names for logical files and Uniform Resource Locators (URLs) to name file instances. Metadata is recorded in a *metadata catalog.* File instances are recorded in a *replica catalog* which maps a logical file name to one or more file instance names. Information about an instance (e.g., the speed of the storage system on which it is stored) is stored in an *instance catalog*, which maps file instance names to such *instance properties.* Logical files may be organized in *collections*. A collection, like a logical file, has a unique name and associated metadata, which in this case includes the names of the collection's constituent logical files.

In this context, a *storage system* is an entity that responds to requests to read and/or write named file instances. Note that the term "storage system" as used here denotes a logical construct and need not map directly to low-level storage. For example, a distributed file system that manages files distributed over multiple storage devices or even sites can serve as a storage system, as can a Storage Resource Broker (SRB) that serves requests by mapping to multiple storage systems of different types. The "names" supported by a particular storage system are also logical constructs, with their meaning being determined solely by the storage system. In many storage systems, a "name" will be a hierarchical directory path, but in some systems (e.g., SRB), it may be a set of attribute-value pairs that the storage system maps internally to an actual file instance.

For conciseness we will use the term *file* to denote what might otherwise be called a "data item" or "entity". This choice of terminology is not intended to imply that Data Grid operations deal only with conventional file systems. On the contrary, a Data Grid implementation might use a system such as SRB to access "file instances" stored within a database management system.

The term *cache* is often used in discussions of Data Grid-like systems, but with widely varying meanings. To reduce confusion, we avoid the use of this term altogether here. Instead, we use the two terms *replication* and *buffering*, as follows:

- Replication denotes the creation of a copy of an entire file instance and the creation of an entry in the replica catalog. In our architecture, replication can be performed under user control or by a library or agent implementing an instance management strategy.
- Buffering denotes the placement of portions of a file instance on faster storage devices. In our architecture, we assume that buffering may be performed within the storage client or storage system implementations but is not under direct user control.

The Data Grid architecture does *not* support user-level management of copies of *partial* files. While experience may eventually motivate the introduction of such support, there are significant advantages in terms of simplicity to dealing with entire files.

1.2 Data Grid Operations

The Data Grid architecture defines a variety of operations on the components just listed, including publication (creation of logical files and collections), instance management (e.g., replication), query mapping (mapping a user query into one or more logical files), instance selection (selecting the appropriate instance of a logical file), data access (application-level read and write operations on all or part of a file instance), and data movement (so-called *storage-to-storage transfers* or *third-party transfers* of a file instance from one storage system to another).

2 API Design

2.1 Design Goals

We note first the principal goals that have motivated the design of the SC-API.

Hide storage system heterogeneity. We wish to hide heterogeneity in storage system architecture as much as possible, by providing a single client-side API that can be used for diverse storage systems, including:

- standard file systems, such as NFS and DFS;
- tertiary storage management systems, such as HPSS;
- servers accessible via Internet protocols, such as HTTP, FTP, and WebDAV; and
- network "caches," such as DPSS.

Provide rich functionality. We desire an API that supports a variety of access methods for remote file instances, including read and write operations on subsets of file instances, read operations that involve the execution of "filters," and third-party transfers in which a file instance is copied from one remote storage system to another. That is, we want more than a least-common denominator set of access methods.

Support high-performance implementations. We wish to allow an implementation to exploit specialized techniques designed to enhance performance and provide performance

guarantees: e.g., network striping, parallel I/O techniques, network protocol tuning, quality of service within storage systems and networks.

Support application-level guidance. We also wish to support application-level performance tuning, via either the provision of hints that can guide lower-level parameter selection or the direct discovery of, and then setting of, low-level parameters.

Leverage Grid infrastructure. We wish to leverage low-level Grid infrastructure such as authentication and instrumentation wherever possible.

2.2 Summary of Storage Client API Operations

The SC-API defines functions for listing the contents of a storage system, for deleting file instances, for opening and closing file instances, for reading and writing contents of file instances, for performing storage-to-storage transfers, and for accessing and setting properties associated with file instances and storage systems. Functions are also provided for manipulating the attribute sets that can be passed to SC-API functions to customize their behavior. In the remainder of this section, we provide an overview of the principal functions of the SC-API. In Section 3, we provide complete function prototypes and definitions for each SC-API operation.

One important property of a storage system is its "scope," that is, the set of machines from which SC-API functions can be used to access it. Many SC-API-enabled storage systems will have global scope, meaning that they can be accessed from any machine in the Internet. However, SC-API functions can also be used to provide access to storage systems that are "local" to a subnet (e.g., a site's NFS file system) or even a single processor (e.g., /tmp). Regardless of where an SC-API function is called from, it is performed on behalf of a specific user, and is subject to local policy constraints (e.g., access control) within the storage system (with identity established via the Grid Security Infrastructure) on which the operation is performed.

The techniques used to implement SC-API functions will depend on both the capabilities of the underlying storage system and on the sophistication of the implementer. For example:

- If a storage system supports appropriate remote partial file read and write operations, then a straightforward implementation approach would translate each SC-API call into the corresponding remote storage operation.
- In the same situation, a more sophisticated implementation might buffer selected file instance contents or property values near the client so as to reduce communication requirements. (We specify below when changes must be committed to the storage system.)
- In the case of a storage system to which remote access is provided only via FTP, an implementation might copy the entire file to an intermediate location on open, then perform read and write operations on that copy.

The following table summarizes the SC-API

Manipulating Storage Systems	**SC-API function**
List	`grid_storage_search`
	`grid_storage_register_search`
Delete	`grid_storage_delete`
	`grid_storage_register_delete`
Open	`grid_storage_open`
	`grid_storage_register_open`
Close	`grid_storage_close`
	`grid_storage_register_close`
Data Access	
Read	`grid_storage_read`
	`grid_storage_register_read`
Write	`grid_storage_write`
	`grid_storage_register_write`
Storage to Storage Transfers	
Transfer data	`grid_storage_transfer`
	`grid_storage_register_transfer`
Manipulating Attribute Sets	
Initialize	`grid_storage_init_attribute_set`
Copy	`grid_storage_copy_attribute_set`
Destroy	`grid_storage_destroy_attribute_set`
Add	`grid_storage_add_attribute`
Remove	`grid_storage_remove_attribute`
Set	`grid_storage_set_attribute`
Get	`grid_storage_get_attribute`
Manipulating Properties	
List	`grid_storage_list_properties`

2.2.1 Manipulating Storage Systems

In the rest of this section, we describe SC-API functionality in more detail. We first introduce the functions for manipulating storage systems. These functions include listing file instances, deleting file instances, and opening and closing file instances.

These operations use Uniform Resource Locators (URLs) to refer to storage systems and file instances. An SC-API URL is, in essence, a concatenation of four components: a protocol name, system name, port number, and name. This convention allows the SC-API to suport the `http`, `ftp`, and certain experimental protocols. The server name and (optional) port number provide the address of the storage system to which requests should be directed.The name is an arbitrary string used by the storage system to identify the file instance. For example:

```
http://myserver.mcs.anl.gov:2000/myfile1
http://myserver.mcs.anl.gov:2000/mydirectory/myfile2
x-srb://srb.sdsc.edu:3000/mycollection&type=grid&name=f
```

The first two examples refer to file instances located within a storage system with server `http://myserver.mcs.anl.gov:2000`. The first instance has local name `myfile1` and the second the hierarchical name `mydirectory/myfile2`. In the third example, we are dealing with a SRB system (with server name `srb.sdsc.edu:3000`) and here the local "name" is a collection name and a set of name-value pairs.

The SC-API provides a function that lists all file instances within a storage system that match a specified search expression. Depending on the nature of the storage system, this function may support directory operations (*e.g.* "list all file instances at a specified point in the directory"). The API also provides a function that deletes named file instances within a storage system.

The SC-API open operation maps from a file instance name (a URL) to a representation that permits direct access to the physical storage: what we call a *file instance handle*. The contents of a handle depend on the underlying storage system. Within a shared file system, a handle might be a file descriptor. In a distributed environment, a handle could be a proxy through which remote read and write requests are forwarded to the remote storage systems, or the handle could be a pointer to a locally buffered copy of the entire file instance. The file instance handle is used for subsequent read and write operations.

If an open operation is performed on a file instance that does not yet exist, the Storage Client API will attempt to create a file instance with the supplied name on the underlying storage system. If this *create on open* operation is successful, a file instance handle is returned.

Finally, the SC-API close operation removes the mapping between the file instance name (URL) and a file instance handle. Changes to file instance properties (such as size and modify times) are not guaranteed to be correct until the close operation completes.

2.2.2 Read and Write Operations

The SC-API's read and write functions provide read and write access to a previously opened file instance. These functions specify a file instance, a starting position within the file instance, a transfer size, and a buffer address.

The behavior that results if multiple readers and writers operate on the same file at the same time is not specified but depends on the semantics of the underlying storage system. That is, we do not require an SC-API implementation to provide stronger coherency guarantees than the underlying storage system. (However, we may decide to associate a property with a storage system that would allow a user to discover what guarantees are provided.)

2.2.3 Storage to Storage Transfers

The SC-API supports storage-to-storage transfers, that is, the direct movement of a file instance from one storage system to another without an intermediate (and often performance limiting) transfer through an application. The SC-API functions in question instruct one storage system to initiate a direct transfer with a second storage system. The initiating system (which may act as either the source or destination for the transfer) then performs the actual transfer via standard SC-API open, close, read, and write calls.

2.2.4 Manipulating Attribute Sets

Because a primary goal of the Data Grid architecture is to support application-level performance tuning, all SC-API storage functions take as an argument an *attribute set*, which the user can employ to pass to the function various parameters, hints, and directives designed to improve data access performance. SC-API functions are provided for creating, modifying, and examining attribute sets. In the prototype implementation, these functions are not thread-safe.

2.2.5 Properties

The SC-API also includes a set of functions for manipulating file instance and storage system properties. *File instance properties* describe individual file instances and include such characteristics as size, owner, access permissions, and last modification date for the file instance. *Storage system properties* describe the storage system and may include the type of storage devices, the default block size, data layout information such as data striping parameters, and a list of interfaces or protocols supported by the storage system, such as FTP, parallel FTP, or MPI/IO. SC-API functions are provided for creating, setting, deleting and listing properties.

Note that the Storage Client API is *not* concerned with *data description properties*, that is, with properties containing semantic information about file instances. A separate metadata access API is used to modify and query these properties.

3 API Specification

We now provide a detailed specification of the various SC-API functions. The various functions have a number of features in common:

- All storage functions take an *attribute set* as an argument. As we explain below, this attribute set is used to pass various parameters, hints and directives used to customize the behavior of the underlying implementation and/or storage system.
- Each function has an asynchronous equivalent if there is any possibility of it blocking. Hence, a user can always write non-blocking code. An asynchronous version of any call always returns immediately. Its argument list extends the argument list of the synchronous version with the callback function that is to be invoked after the operation completes, a pointer to any arguments for that callback function, and an optional pointer to a callback handle. The callback handle can be used to cancel an outstanding grid storage operation.
- All functions use the *Grid error handling API, which is defined in a separate document*. This means that their return value is a handle (of type `globus_result_t`) which can either be ignored or used to access an error object. A handle value of GLOBUS_SUCCESS indicates success; other values indicate errors, in which case the application can use the handle to retrieve an error object that contains information about the type of error that occurred.

3.1 Manipulating Storage Systems

3.1.1 List File Instances in a Storage System

```
globus_result_t
grid_storage_search(
   grid_storage_url_t *       storage_url,
   char *                     search_expression,
   grid_storage_url_t **      url_array,
   grid_storage_attribute_t * attributes)


globus_result_t
grid_storage_register_search(
   grid_storage_url_t *       storage_url,
   char *                     search_expression,
   grid_storage_url_t **      url_array,
   grid_storage_attribute_t * attributes,
   grid_storage_search_callback_t
                              callback_fn,
   void *                     callback_args,
   grid_storage_callback_handle_t *
                              callback_handle)
```

Places in `url_array` a NULL terminated sequence of URLs for all file instances in the storage system named by `storage_url` that match the supplied `search_expression`.

3.1.2 Delete a File Instance

```
globus_result_t
grid_storage_delete(
   grid_storage_url_t *       storage_url,
   grid_storage_attribute_t * attributes)

globus_result_t
grid_storage_register_delete(
   grid_storage_url_t *       storage_url,
   grid_storage_attribute_t * attributes,
   grid_storage_callback_t    callback_fn,
   void *                     callback_args,
   grid_storage_callback_handle_t *
                              callback_handle)
```

Instructs the underlying storage system to delete the file instance named by `storage_url`. Any operations on active handles referring to this URL may fail after the URL is deleted.

> The original text for this operation was:
>
> *Removes mapping between URL, handle, and physical storage. If there are no other mappings to this physical storage location, instruct the underlying storage system to delete the file instance.*
>
> The "mapping" concept is not defined in this document. The behavior of using this API is undefined if multiple concurrent accesses to the URL are occurring. A handle may or may not become invalid immediately upon file deletion. For example, an HTTP server will not know that a client has an open handle to an URL, because each URL GET or PUT operation is stateless.

3.1.3 Open a File Instance

```
globus_result_t
grid_storage_open(
   grid_storage_url_t *       item_url,
   grid_storage_handle_t *    item_handle,
   grid_storage_attribute_t * attributes)

globus_result_t
```

```
grid_storage_register_open(
    grid_storage_url_t *        item_url,
    grid_storage_handle_t *     item_handle,
    grid_storage_attribute_t *  attributes,
    grid_storage_callback_t     callback_fn,
    void *                      callback_args)
```

Opens the file instance named by `item_url` and returns as `item_handle` a handle that can be used to perform subsequent read and write operations on that file instance. This handle is not required to be valid outside the scope of the requesting process, hence it cannot be passed to, or inherited by, other processes.

The `attributes` supplied when making this call are used to set properties of the opened file instance, such as the open mode (read-only, read-write, etc), and whether to create the URL if it does not exist yet.

In the prototype implementation, the following attributes are defined for the open calls:

GRID_STORAGE_ATTRIBUTE_OPEN_FLAGS. The value of this attribute should be a value string representation of the integer value passed as the second argument to the POSIX `open`() call.

GRID_STORAGE_ATTRIBUTE_CREATE_MODE. The value of this attribute should be a string representation of the integer value passed as the third argument to the POSIX `open`() call.

The original description for this API function contained a statement to the effect that a file instance would be created on open, if it did not exist. In the prototype implementation, file instance creation will only happen if the GRID_STORAGE_ATTRIBUTE_OPEN_FLAGS attribute value is the logical OR of O_CREAT.

The "create on open" operation does *not* modify metadata or replica catalogs, for example to record a mapping between a logical file name (URN) and the name (URL) of the newly created file instance. This mapping must be established separately by using the metadata and replica manager APIs.

Q: How to deal with DPSS, which requires specific number of bytes for create operation?

We need to think further about access control issues, as a user creating a file will want to be able to control who can access it.

Errors that can occur during the open operation include the following:

- Request Errors (Client may change request and retry):
    - Authentication of client fails

- Authorization of client fails (not allowed to open file instance in specified mode)
- File instance unknown: file not on storage system
- Storage system unknown: can't contact storage system

- Communication Failures:
  - Network or storage system unreachable
  - Protocol mismatch
  - Protocol violation
- Transfer Aborted
  - By user
  - By remote storage system
  - Due to internal error

3.1.4 Close a File Handle

```
globus_result_t
grid_storage_close(
   grid_storage_handle_t *    item_handle,
   grid_storage_attribute_t * attributes)

globus_result_t
grid_storage_register_close(
   grid_storage_handle_t *          item_handle,
   grid_storage_attribute_t *       attributes,
   grid_storage_callback_t          callback_fn,
   void *                           callback_args)
```

Close the file instance specified by the supplied `item_handle`. Changes to file instance properties are not guaranteed to be correct until the file instance handle is closed. For example, as clients write to an open file instance, its size property changes to reflect these writes. Properties such as size may change while the file instance is open, but they are not guaranteed to be correct until the file instance is closed.

In the prototype implementation, the properties of an URL are read-only to the application. I'm not sure if the statement above this box has any meaning right now.

Is there a situation where one would wish to close a file instance, without having access to the handle? For example, what if a program crashes and we've lost the handle?

3.2 Read and Write Operations on File Instances

3.2.1 Read from a File Instance

```
globus_result_t
grid_storage_read(
   grid_storage_handle_t*     item_handle,
   unsigned char *            buffer,
   size_t                     offset,
   size_t                     bufsize,
   grid_storage_attribute_t * attributes)

globus_result_t
grid_storage_register_read(
   grid_storage_handle_t*     item_handle,
   unsigned char *            buffer,
   size_t                     offset,
   size_t                     bufsize,
   grid_storage_attribute_t * attributes,
   grid_storage_callback_t    callback_fn,
   void *                     callback_args)
```

Synchronous read from a file instance handle. Unlike standard UNIX read operations, storage system read calls do not assume a file pointer. This reduces the amount of state the server and client must maintain. Instead, each read call specifies an offset within the file instance and the size of the buffer that will hold read data. The attributes data structure can be used to pass extensible arguments or hints to the storage system, such as block size. Eventually, we expect to also allow the set of valid read attributes to include a filter to run on the data. Attributes can also be used to specify aggregate read calls.

In addition to the errors specified for the open operation, the following errors can occur on read operations:

- Out of Range: Offset and/or size require read past end of file instance

3.2.2 Write to a File Instance

```
globus_result_t
grid_storage_write(
   grid_storage_handle_t*     item_handle,
   unsigned char *            buffer,
```

```
    size_t                      offset,
    size_t                      bufsize,
    grid_storage_attribute_t *  attributes)

globus_result_t
grid_storage_register_write(
    grid_storage_handle_t *     item_handle,
    unsigned char *             buffer,
    size_t                      offset,
    size_t                      bufsize,
    grid_storage_attribute_t *  attributes,
    grid_storage_callback_t     callback,
    void *                      user_arg)
```

As for read, the write call does not assume a file instance pointer, but instead specifies the position within the file instance and the size of the write operation.

The attributes data structure holds extensible arguments or hints to the storage system, such as block size.

Side effects of the write operation include changing the size and modify properties of the file instance. Although these changes may take place while the logical file is open, the properties are not guaranteed to be correct until the file instance handle is closed. This is a departure from standard UNIX semantics, which requires size attributes to be updated as write operations complete.

3.3 Storage-to-Storage Transfers

The functions described above (e.g. open, read, write, etc) allow an application to access subsets of a file instance on a particular storage system. We additionally need the ability to transfer entire file instances efficiently from one storage system to another.

An application can perform such a transfer simply by using the read, write, and other calls defined above. However, this approach may yield poor performance, as all data is transferred through the application.

Instead, it is often preferable to perform a so-called “storage-to-storage transfer” in which two storage systems transfer data directly between them. Such storage to storage transfer has several potential performance advantages. It does not require all data to pass through a single client, which can be a bottleneck. Also, if a storage system is actually composed of multiple underlying devices (e.g., multiple disk and/or tape drives), the transfer can be performed by (or close to) the underlying devices, thus avoiding intermediate copies, transfers, and/or layers of software.

3.3.1 Storage-to-Storage Transfer

```
globus_result_t
grid_storage_transfer (
    grid_storage_url_t *            source_url,
    grid_storage_url_t *            destination_url,
    grid_storage_attribute_t *      attributes)

globus_result_t
grid_storage_register_transfer (
   grid_storage_url_t *             source_url,
   grid_storage_url_t *             destination_url,
   grid_storage_attribute_t *       attributes,
   grid_storage_callback_t          callback_fn,
   void *                           callback_args)
```

Perform a storage-to-storage transfer of the file instance specified by `source_url` to the file instance specified by `destination_url`, creating the destination file if it does not already exist. An implementation may choose to perform the transfer as a push from the source storage system, a pull from the destination storage system, or via a series of read and write operations from the client or some intermediate location, depending on the properties of the various resources involved. (See implementation notes below.) The user may use attributes to express a preference for one implementation approach or another.

3.3.2 Storage-to-Storage Implementation Approaches

The storage-to-storage functions defined here do not specify a particular implementation approach; as noted in the function description, an implementation can (and should) choose between one of several alternative strategies, depending on the capabilities of the source and destination storage systems (and perhaps guided by user-supplied attributes). Obvious strategies to consider include the following:

1. We can initiate a “push” of the file instance from the source storage system to the destination storage system. This strategy requires the ability to initiate the “push” operation at the source storage system, which may or may not be possible.
2. We can initiate a “pull” of the file instance at the destination storage system, from the source storage system. This strategy requires the ability to initiate the “pull” operation at the destination storage system, which may or may not be possible.
3. We can perform the transfer via a series of SC-API read and write operations within the client. This strategy is always possible but, as noted above, can have performance problems.

4. We can perform the transfer via a series of SC-API read and write operations at some intermediate location. This strategy may be useful when strategies (1) and (2) are not possible, and strategy (3) is too slow due to poor client performance.

Strategies (1) and (2) require that the two storage systems speak a common protocol. This requirement can be satisfied in two different ways:

1. We can define a single common transfer protocol that will be used by all storage systems. This protocol may be some existing protocol (e.g., ftp), or a new protocol. While such an implementation is conceptually simple, in practice it is not feasible to require every storage system to implement a particular transfer protocol since some storage systems cannot be modified (e.g., commercial HTTP servers). Further, it is a significant research effort to define a perfect storage to storage transfer protocol.
2. One storage system can perform the transfer using the SC-API data access interface of the other storage system. The storage system performing the transfer will use standard SC-API functions (open, read, write, close) to read from or write to the other storage system. Advantages of this approach include the use of standard protocols and APIs and support for multiple protocols.

We believe that an SC-API implementation should adopt the second of these two options.

3.4 Flush Operations

We need to add a section here specifying the types of flush operations that we support (e.g., flush to the network, flush to the remote storage system). We will also specify what errors are guaranteed not to occur after a flush operation completes.

We also need to add a corresponding high level description in Section 2.

3.5 Manipulating Attribute Sets

The SC-API calls specified above all include an *attribute set* argument. An attribute set contains zero or more (attribute-name, attribute-value) pairs, where attribute names and values are strings. An attribute set is used to pass various parameters, hints and directives used to customize the behavior of the underlying implementation and/or storage system. For example, attributes might provide hints about how to perform a read operation or how to set the values of storage system properties.

In this section, we define functions for creating attribute sets, for modifying the values associated with attributes contained within an attribute set, and for deleting attributes from an attribute set.

Note: The simple string manipulation functions described here are intended for use only in the initial SC-API prototype. We plan a very different implementation of attributes for the final version. For each SC-API function (open, close, read, write, etc.),

we will define a standard set of possible attributes. Then we will implement function calls for manipulating each set of attributes, including init, destroy, set and get.

3.5.1 Initialize an Attribute Set

```
globus_result_t
grid_storage_init_attribute_set(
   grid_storage_attribute_t * attribute_set)
```

Create a new attribute set. The `attribute_set` argument will be initialized to be an empty attribute set.

Errors that can occur during the init operation are:

- GRID_STORAGE_ERROR_NULL_PARAMETER

  An attempt to initialize a NULL `attribute_set`.

3.5.2 Copy an Attribute Set

```
globus_result_t
grid_storage_copy_attribute_set(
   grid_storage_attribute_t * attributes,
   grid_storage_attribute_t * copy)
```

Copy the contents of the attribute set `attributes` to the new attribute set copy. The `copy` attribute set is assumed to be uninitialized.

3.5.3 Destroy An Attribute Set

```
globus_result_t
grid_storage_destroy_attribute_set(
   grid_storage_attribute_t * attributes)
```

Destroy the attribute set `attributes`. All name/value pairs in the attribute set will be removed, and the attribute set will be destroyed. Any outstanding references to attribute values obtained via `grid_storage_get_attribute` are invalid, and should no longer be used.

3.5.4 Add an Attribute

```
globus_result_t
grid_storage_add_attribute(
   char *                    attribute_name,
   char *                    attribute_value,
```

```
    grid_storage_attribute_t *  attribute_set)
```

Adds the attribute (`attribute_name`, `attribute_value`) to the supplied `attribute_set`. The operation fails if the named attribute already exists within the supplied `attribute_set` or if space for the new attribute cannot be allocated.

Errors that can occur during an add operation are:

- GRID_STORAGE_ERROR_NULL_PARAMETER

  An attempt to add a NULL `attribute_name`, or an attempt to add the attribute to a NULL `attribute_set`.

- GRID_STORAGE_ERROR_OUT_OF_MEMORY

  Unable to allocate memory to store the (`attribute_name`, `attribute_value`) pair in the `attribute_set`.

- GRID_STORAGE_ERROR_ALREADY_EXISTS

  An attempt to add an `attribute_name` which is already defined in the `attribute_set`.

3.5.5 Remove an Attribute from an Attribute Set

```
globus_result_t
grid_storage_remove_attribute(
   char *                      attribute_name,
   grid_storage_attribute_t *  attribute_set)
```

Remove the attribute named attribute_name from the attribute set `attribute_set` to `attribute_value`. Any references to the attribute's value obtained from `grid_storage_get_attribute` are no longer valid, and should not be referenced.

Errors that can occur during a set operation are:

- GRID_STORAGE_ERROR_NULL_PARAMETER

  An attempt to remove a NULL `attribute_name`, or an attempt to remove the attribute from a NULL `attribute_set`.

- GRID_STORAGE_ERROR_DOES_NOT_EXISTS

  An attempt to remove an `attribute_name` which is not defined in the `attribute_set`.

3.5.6 Set an Attribute

```
globus_result_t
grid_storage_set_attribute(
    char *                      attribute_name,
    char *                      attribute_value,
    grid_storage_attribute_t *  attribute_set)
```

Sets the value associated with the attribute named `attribute_name` within `attribute_set` to `attribute_value`, or fails if no such attribute exists within the supplied attribute set.

Errors that can occur during a set operation are:

- GRID_STORAGE_ERROR_NULL_PARAMETER

  An attempt to set a NULL `attribute_name`, or an attempt to set the attribute in a NULL `attribute_set`.

- GRID_STORAGE_ERROR_OUT_OF_MEMORY

  Unable to allocate memory to store the (`attribute_name`, `attribute_value`) pair in the `attribute_set`.

- GRID_STORAGE_ERROR_DOES_NOT_EXISTS

  An attempt to set an `attribute_name` which is not defined in the `attribute_set`.

3.5.7 Get an Attribute's value

```
globus_result_t
grid_storage_get_attribute(
    char *                      attribute_name,
    char **                     attribute_value,
    grid_storage_attribute_t *  attribute_set)
```

Retrieve the value of an attribute with name `attribute_name` from `attribute_set`, or fails if no such attribute exists within the supplied attribute set. If successful, the `attribute_value` parameter will point to a string containing the value of the attribute. This string should not be modified or freed by the user.

Errors that can occur during a delete operation are:

- GRID_STORAGE_ERROR_NULL_PARAMETER

  An attempt to delete a NULL `attribute_name`, or an attempt to delete the attribute to a NULL `attribute_set`.

- GRID_STORAGE_ERROR_DOES_NOT_EXISTS

  An attempt to delete an `attribute_name` which is not defined in the `attribute_set`.

3.6 Manipulating File Instance and Storage System Properties

The functions defined here allow users or applications to set and query properties of file instances or storage systems. We define an interface similar to the attribute interface for our prototype implementation.

Like the attribute interface above, the specification of the property management functions is likely to change dramatically in subsequent versions of SC-API.

We first define sets of required properties for file instances and storage systems.

Required properties for file instances include:

- type
- access mode
- owner
- modify time
- size
- reference count

Required storage system properties:

- storage device type
- device properties (e.g., for disks: sectors per track, tracks per cylinder, revolutions per second, rotational delay between contiguous blocks, etc.)
- block size
- layout and allocation policies (e.g., striping)
- network protocols supported (e.g., MPI/IO, ftp, pftp)
- authentication scheme

Is it ok to let applications set these properties directly? Or should they only be set as side effects of Storage Client API calls (i.e., open, write, close)?

3.6.1 List the properties of an URL

```
globus_result_t
```

```
grid_storage_list_properties(
    grid_storage_url_t *            storage_url,
    grid_storage_attribute_t *      attributes,
    char ***                        property_names,
    char ***                        property_values)
```

Return a list of the properties of the file instance or storage system. The property_names and property_values arrays will be initialized to a NULL-terminated copy of the properties of the URL. The user must free the property name and property value arrays.